# Formalization of software architectures to study the preservation of properties in composition of components: an example using category theory in the context of Big Data
CAiSE 2021

Annabelle Gillet, Éric Leclercq and Nadine Cullot

March 2021

## 1 Introduction

When architectures evolve and grow, they can combine several smaller parts of architectures developed separately. When building a large scale, complex and distributed architecture, its parts can embed architecture styles on their own. These different cases can result in compositions of smaller architecture parts with their own properties, so formalization should be able to express and control these compositions.

To fill this need, category theory [EM45] is a promising approach: it allows to switch from a model to another or to navigate among abstraction levels, and thus to express various problems from different science fields, such as mathematics, physics or computer science [Spi14, FS19]. By focusing on relations (the morphisms) and compositions, it proposes powerful mechanisms that can be applied to architectures. We focus on studying the conservation or the discarding of properties in compositions of components, by relying on the behaviour of functors combined to preorders. With this formalization, it is possible to deduce:

1. the value for a complex property by knowing values of simple properties that they rely on ;
2. the value of a property for a composition of components by knowing the value taken individually by each component of the composition.

## 2 Essential notions of category theory

This section introduces some basics notions of category theory useful to understand the study of properties in compositions of components: the categories and the functors.

**Definition 2.1.** Category
A **category** $C$ is composed of four fundamentals elements:

1. $\mathrm{Ob}(C)$, a collection of objects ;
2. for each pair $x, y \in \mathrm{Ob}(C)$, a set $\mathrm{Hom}_C(x, y)$ representing **morphisms** from $x$ to $y$, namely a mean to get an object $y$ (the codomain) from an object $x$ (the domain). A morphism $f$ from $x$ to $y$ is noted $f : x \to y$ ;
3. for each $x \in \mathrm{Ob}(C)$, a particular morphism $\mathrm{id}_x$ known as the identity morphism on $x$ ;
4. for each triplet $x, y, z \in \mathrm{Ob}(C)$, a **composition** $\circ : \mathrm{Hom}_C(y, z) \times \mathrm{Hom}_C(x, y) \to \mathrm{Hom}_C(x, z)$. For two morphisms $f : x \to y$ and $g : y \to z$, the composition is noted $g \circ f : x \to z$.

And of two laws:

1. for a morphism $f : x \to y$ with $x, y \in \mathrm{Ob}(C)$, we have $f \circ \mathrm{id}_x = f$ and $\mathrm{id}_y \circ f = f$ ;
2. for $f : w \to x$, $g : x \to y$ and $h : y \to z$ with $w, x, y, z \in \mathrm{Ob}(C)$, we have $(h \circ g) \circ f = h \circ (g \circ f) \in \mathrm{Hom}_C(w, z)$.

In diagrams, categories are represented by boxes. Functors are represented by thick arrows and their effect on objects by dashed arrows. To simplify, identity morphisms are not represented, even if they are always present.

**Definition 2.2.** Functor

A **functor** $F$ is a morphism between two categories. It maps a category $C$ to a category $C'$. It is noted $F : C \to C'$, and acts on:

1. objects: for each object $x \in \mathrm{Ob}(C)$, we have an object $F(x) \in \mathrm{Ob}(C')$ ;
2. morphisms: for each pair $x, y \in \mathrm{Ob}(C)$, we have $F : \mathrm{Hom}_C(x, y) \to \mathrm{Hom}_{C'}(F(x), F(y))$.

A functor must follow two laws to ba valid:

1. the preservation of identities: $\forall x \in \mathrm{Ob}(C)$, $F(\mathrm{id}_x) = \mathrm{id}_{F(x)}$ ;
2. the preservation of composition: for any triplet $x, y, z \in \mathrm{Ob}(C)$ with morphisms $g : x \to y$, $h : y \to z$, we have $F(h \circ g) = F(h) \circ F(g)$.

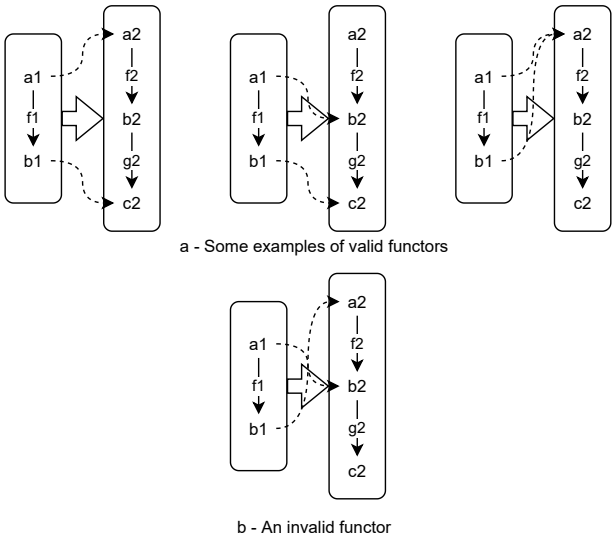


Figure 1: Example of the law of preservation of compositions ($F(h \circ g) = F(h) \circ F(g)$)

The effect of functors on morphisms enforces strong compelings for the validity of the functor. These constraints can be used to deduce several properties. For each object linked to another category with a functor, the functor must also preserve its morphisms. Figure 1 presents some examples of the application of a functor $F$ from a category $C1$ to a category $C2$. In the top subfigure, all the functors are valid: the first transforms objects with $F(a1) = a2$ and $F(b1) = c2$, and the morphism $f1$ is preserved with $F(f1) = g2 \circ f2$. The second is direct and transforms objects with $F(a1) = b2$ and $F(b1) = c2$, and the morphism $f1$ with $F(f1) = g2$. The third relies on the identity morphisms to be valid, and transforms objects with $F(a1) = a2$ and $F(b1) = a2$, and the mrophism $f1$ with $F(f1) = id_{a2}$. The bottom subfigure shows an invalid functor: if this functor transforms objects with $F(a1) = b2$ and $F(b1) = a2$, then there is no mrophism in $C2$ that goes from $b2$ to $a2$ to validate the functor.

# 3 Advanced definitions

Some more advanced concepts of category theory needed to formalize architectures are presented in this section.

**Definition 3.1.** Preordrer

**Preorders** are a special type of category, in which between each pair $x, y \in \text{Ob}(C)$, there exists a unique morphism $f : x \to y$. If there exist $f : x \to y$ and $g : x \to y$, then $f = g$.

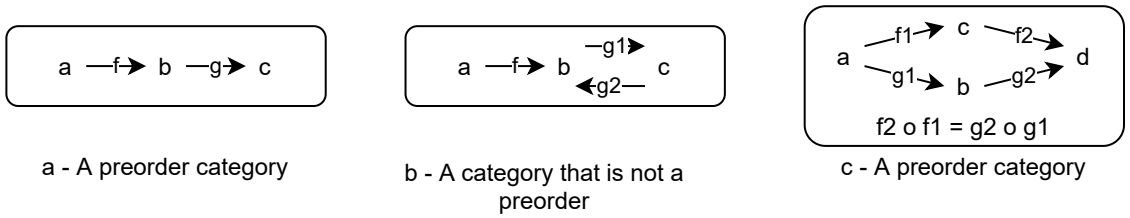


Figure 2: Illustration of preorders

Preorders provide a form of hierarchy in a category. Indeed, for each pair $x, y \in \text{Ob}(C)$, there can be only one morphism, including compositions. The part a of the figure 2 shows a preorder, because there is only one morphism between each pair of objects. The part b is not a preorder, because to get $b$ from $a$, it is possible to use the morphism $f$ or the composition $g_2 \circ g_1 \circ f$. Thus, preorders exclude cycles of morphisms, as it provides many ways to get an object from another object. Nonetheless, it is possible to have multiple morphisms that link a pair of objects in a preorder, at the condition of specifying their equality in an equation, as illustrate in the part c of the figure. In this document, we will omit the equation to simplify figures.

**Definition 3.2.** Power set

The **power sets**, that are sets which contain all the subsets of a given set. In a category, power sets can be organized as preorder, where subsets are objects and morphisms link two subsets if a subset $x$ is integrally included in a subset $y$.

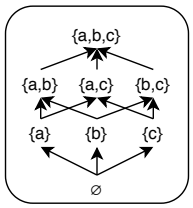


Figure 3: A **power set** for the set $\{a, b, c\}$

An example of power set if given in the figue 3.

**Definition 3.3.** Product of categories

A **product** of two categories is an operation that produces a new cateogry from two existing categories $C1$ and $C2$. The objects of this new category are all the possible pairs $(x, y)$ with $x \in \text{Ob}(C1)$ and $y \in \text{Ob}(C2)$ and the morphisms $(x, y) \to (x', y')$ are pairs $(f, g)$ where $f : x \to x' \in \text{Hom}_{C1}(x, x')$ and $g : y \to y' \in \text{Hom}_{C2}(y, y')$.

Figure 4 shows a product of categories. Part a presents how objects and morphisms are created in the new category. Part b demonstrates how a functor can be applied on the result of a product of categories. Part c shows the diagram simplification of the part b that we will use in the rest of this document to simplify

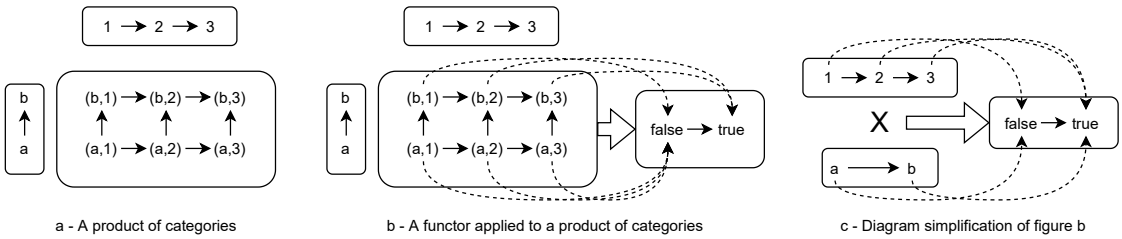


Figure 4: Product of categories and its diagram simplification

diagrams, only when it is enough to represent the effect of the functor. This simplification is valid when a functor is applied on the result of a product of categories between two preorders, and that the category codomain of the functor is also a preorder. It can be used to deduce the effect of the functor on a pair of objects of the product of categories knowing only the targeted object in the new category by each individual object of the pair: the result will be the lowest targeted object in the new category. In our example, if the object $(a, 3)$ would have been transformed in the object *true* in the new category, the diagram simplification could not have been used.

# 4 Principles of the formalization

To formalize an architecture, we use three predefined categories and two predefined functors:

**Definition 4.1.** The category *Components*
In this category, the objects are the components of the architecture, without morphisms to link them, except the identity morphisms.

**Definition 4.2.** The category *Architecture*
In this category, the objects are the components, that are linked by a morphism to represent the interactions. A morphism $f : x \to y$ implies that the component $x$ sends data to the component $y$.

**Definition 4.3.** The category *ComponentsPS*
In this category, the objects are the subsets of the power set of the components.

The category *ComponentsPS* will be used to link components to properties, and to study the conservation in the compositions of components.

**Definition 4.4.** The functor $CA$
The functor $CA$ is defined between the categories *Components* and *Architecture* ($CA : Components \to Architecture$). It links individual components to their position in the architecture. Thus, this functor is used to integrate components in the whole architecture.

**Definition 4.5.** The functor $CCPS$
The functor $CCPS$ is defined between the categories *Components* and *ComponentsPS* ($CCPS : Components \to ComponentsPS$). It links individual components to subsets of the power set with only one element. In this manner, it is possible to study the behaviour of properties in compositions of components.

The figure 5 shows the formalization of a simple architecture with two components $a$ and $b$, represented in the category *Components*. The category *Architecture* is used to define that $a$ sends data to $b$. Each component is sent to its subset in the category *ComponentsPS*. Morphisms of the *ComponentsPS*-category gather components in compositions.

This formalization allows the study of the conservation of properties in compositions of components, by relying on the definition of functors, power sets and products of categories. Functors include in their

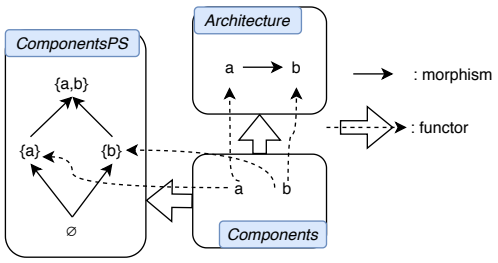


Figure 5: A simple architecture with two components $a$ and $b$

structural definition the notion of composition. Therefore their simple definition, if it is syntactically valid, is sufficient to prove the preservation or loss of a property.

**Definition 4.6.** Formalization of a property

A property is represented by a preorder, in which objects are the different values of the property and morphisms go from the most satisfying value to the least satisfying. The symbol $\top$ is used for component that are not concerned by a property, to avoid impacting compositions of components. $\top$ is the domain of only one morphism, with the most satisfying value as codomain.

Properties can be simple (only with $true$ and $false$ objects), multivalued, or more complex, and resulting of the composition of several other properties: in this case, properties are associated with a product of categories, and this product is linked to the next property with a functor that maps each combination to its signification in the next level property. The complexity of properties can be identified by giving a level to each property: those linked directly to the components of the architecture are level 1, and those depending on product of categories of a higher level. The category $ComponentsPS$ is connected to every property of level one with functors.

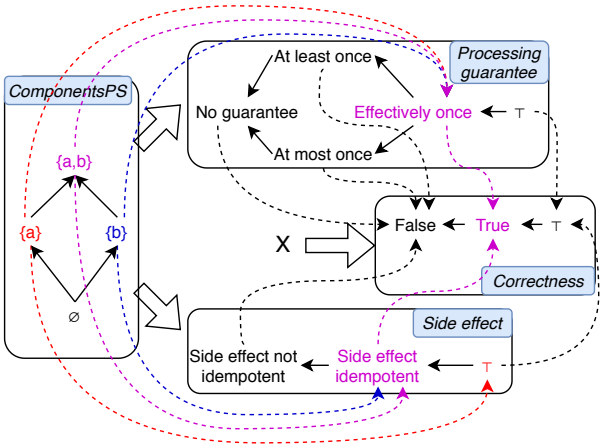


Figure 6: Deduction of a complex property through product of categories

The example depicted in figure 6 studies how the correctness property is supported in an architecture with two components, $a$ and $b$, that use stream processing. The correctness depends on the processing guarantee, as well as the presence or absence of side effects, and if those side effects are idempotent or not. To simplify the schema, the product between the two level one properties is not directly drawn, and is symbolized by a X. Using the power of category theory, we can extract some high-level knowledge from the former representation:

1. by respect to the second property of functors (i.e. $F(h \circ g) = F(h) \circ F(g)$), if a component reaches a lower value for a given property than other components, and if this component is added to the

others, the new set of components will reach the lowest value of those taken by the components for this property. **This is essential in the formalization, as it highlights which composition of components leads to the loss of a property**. In this example, the component $a$ reaches the $\top$ value in the $Side - effect$-category and $b$ reaches the side effect idempotent value, thus, the overall architecture takes the lowest value: the side effect idempotent one ;

2. the product of two level one categories allows to get to a level two category automatically depending on the functor defined between the product and the level two property. In our example, $\{a, b\}$ (the overall architecture), is sent to the value effectively once in the $Processing - guarantee$-category, and to the value side effect idempotent in the $Side - effect$-category. The lowest value reached by the components in the $Correctness$-category is true, thus the correctness property is effective in the architecture.

The definition of functors, and more particularly $F : \mathrm{Hom}_C(x, y) \rightarrow \mathrm{Hom}_{C'}(F(x), F(y))$, allows to deduce the value of a property for a composition of components: **the loss of a property by a component cannot be compensate**. Furthermore, for a complex property, all the functors that map a product of categories to a property with a higher level are defined beforehand. They allow to automatically deduce the value of a complex property for a component or a composition of components, knowing only their value for the level 1 properties.

# 5 Application of the formalization

This formalization allows us to precisely compare the patterns of the Lambda Architecture and the Lambda+ Architecture. To simplify diagrams in this section, the $ComponentsPS$-category has only individual components, as well as possible running compositions of the architecture, and only the links from components to properties that do not lead to $\top$ are represented.

## 5.1 The Lambda Architecture

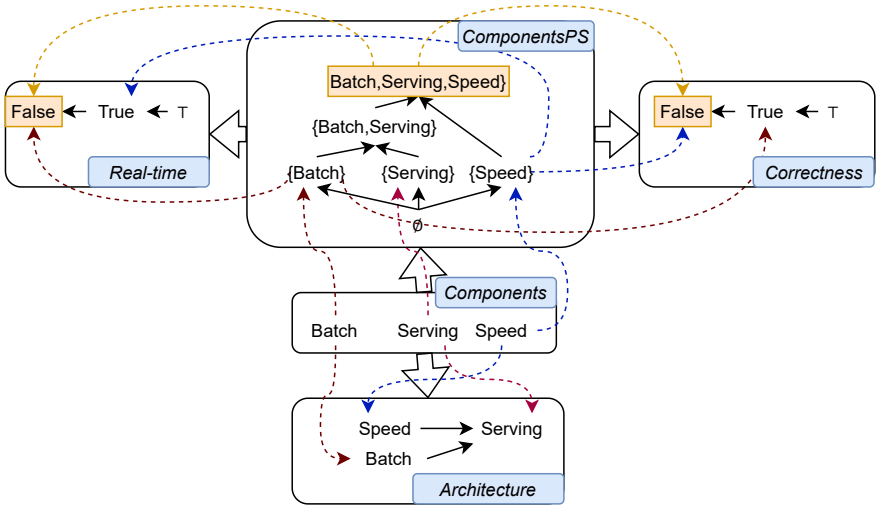


Figure 7: Formalization of the Lambda Architecture pattern

The figure 7 shows the formalization of the Lambda Architecture pattern. The serving layer is neutral concerning the correctness and the real-time properties, as it only puts data that have been computed in the other layers to disposal. The speed layer has the real-time property, but not the correctness, and the

batch layer is the opposite, with the correctness property but not the real-time. By supporting the real-time property only in the speed layer, and the correctness property only in the batch layer, the running composition of components does not support these properties simultaneously. It results in an architecture that produces incorrect results continuously, and corrects it periodically in a batch fashion.

### 5.2 The Lambda+ Architecture

The figure 8 shows the formalization of the Lambda+ Architecture pattern. The data traffic controller Two running compositions of components are possible: with or without the master dataset, as it is only linked to the streaming ETL when data have to be reprocessed. Contrary to the Lambda Architecture, the real-time property is only inactive when data are extracted from the master dataset, but in exchange the Lambda+ benefits from the fault-tolerance property. It integrates also the advances of stream processing systems, thus leveraging the correctness property as long as the side effects are idempotent (if they are present), and that the effectively-once property is available.

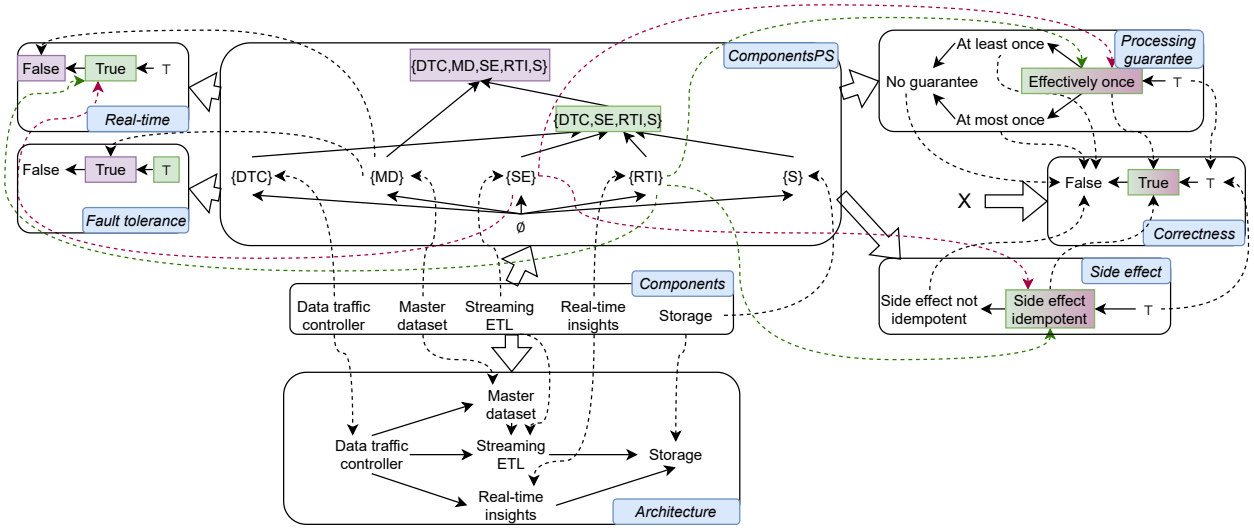


Figure 8: Formalization of the Lambda+ Architecture pattern

## References


[EM45] Samuel Eilenberg and Saunders MacLane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, 58(2):231–294, 1945.

[FS19] Brendan Fong and David I Spivak. *An invitation to applied category theory: seven sketches in compositionality*. Cambridge University Press, 2019.

[Spi14] David I Spivak. *Category theory for the sciences*. MIT Press, 2014.